open proceedings

Optimizing Goodput through Sharing for Batch Analytics with Deadlines

Srinivas Karthik[1†] Panagiotis Sioulas[2‡] Ahana Pradhan[1]

Raghunandan Subramanya[1] Ioannis Mytilinis[2 ‡] Anastasia Ailamaki[3]

1: Independent, India 2: Oracle, Switzerland 3: EPFL & RAW Labs, Switzerland
1: {skarthikv, raghunandan.subramanya, sendatahana}@gmail.com, 2: {firstname.lastname}@oracle.com,
3:{firstname.lastname}@epfl.ch

ABSTRACT

Modern big data systems process not only a huge volume of data, but also numerous concurrent queries. These queries usually span over distributed data and need to be processed within a strict deadline (e.g.: report generation, SLA, etc.). While users expect all queries to be completed within these tight deadlines, the presence of failures (causing delays) often leads to relying on best-effort solutions. Commonly, this involves maximizing the number of queries that can be completed within the batch before the deadline.

In order to address this issue, existing systems typically aim to maximize system's throughput either by enchancing single-query performance or by sharing work, while being oblivious to deadline. Previous studies on real-time systems have proposed methods centered around meeting deadlines, but miss opportunities resulting from overlap of work among queries in the batch. Thus, these limitations result in providing limited performance.

In this paper, we present a novel system called BIGSHARED, which aims to maximize queries that complete by harnessing the advantages of reusing computations through work-sharing. We introduce a unique *deadline-conscious batch optimizer* that manages the delicate balance between meeting deadlines and leveraging sharing. To further improve performance, especially in the event of failures, we incorporate an efficient fault-tolerance mechanism through *sharing-conscious checkpointing*. We evaluate the performance of BIGSHARED using queries from the TPCDS benchmark on an open source big data system. The experimental results demonstrate that, on average, BIGSHARED surpasses the optimal query-at-a-time approach by 61%, and outperforms a state-of-the-art work-sharing system by 74%, thus showcasing significant benefits of amalgamating deadline-aware and sharing-aware paradigms.

1 INTRODUCTION

Businesses collect and analyze data from various sources such as customer feedback, sales metrics, financial records, and market trends to gain valuable insights into their operations. By using this data, executives and managers can better understand the strengths and weaknesses of their business, identify areas that require improvement, and devise strategies to optimize performance. It is of paramount importance in current times with fast-paced industries where making decisions in a timely manner is essential for achieving business success.

To accomplish faster insights and enable better decisions, data management systems must be capable of processing large batches of queries with short deadlines, which can place significant stress on the system. An important hurdle big systems face while meeting the deadline is due to unintended or recurrent failures in the operating environment. These failures induce delays in query response times, as they may typically need to redo parts or the entire set of queries. This results in users resorting to best effort solutions w.r.t. query results. Out of the many ways of relaxing the requirements, a standard way is to maximize the number of queries that finish before the deadline, if not all.

Example use-case: Our motivation for this work comes from analytical use-cases, exemplified by a stock trading scenario. In this context, batches of analytical queries are executed to generate comprehensive stock information within strict deadlines, aiding prompt decisions like identifying stocks to buy within the next hour. As insights accumulate, new query sets are devised to delve deeper in subsequent intervals. Failures occasionally impede query completion before deadlines, prompting decision-makers to seek maximal extractable information.

Prior-work: Big data systems like Spark [1], Trino [2], Greenplum [3], and Databricks [4] are the go-to solutions for handling large batches of queries with deadlines. They aim to improve the response time of individual queries, and hence result in more queries finishing before the deadline. The limitation of this approach is to process queries individually, which fails to take advantage of opportunities to save processing time through the commonality of work across a batch. Jindal et al. [5] illustrate this point by showing that up to 45% of jobs in Microsoft production workloads are similar or repeated, resulting in redundant costs that can amount to millions of dollars when using the query-at-a-time model.

To overcome the aforementioned limitation, *work sharing systems* have been developed, which are designed to holistically improve batch response time by reusing computation. These systems have gained attention for their ability to share work, which can be achieved through various formulations, such as view selection, multi-query optimization, subexpression reuse, and shared operators. These systems have been studied extensively in the literature, with examples including [6–9].

Goodput: Overall, both the above lines of work have focused on enhancing a system's throughput. However, this approach has its limitations as it measures the amount of work processed in a

† Majority of the work was done while the author was at EPFL
‡ The work was done while the authors were at EPFL

given time-frame without taking into account whether that work is useful or not. Whereas goodput metric measures the amount of useful work completed within a set time limit, specifically the number of queries completed per unit time for a predefined interval. Therefore, to achieve optimal performance within time limits, a system should prioritize maximizing its goodput.

To ensure deadlines are met (or maximize goodput), many techniques have been proposed in the past in the context of real-time systems, wherein the queries are rescheduled based on the use-case. For example, these batch queries are rescheduled as per the shortest job first order [10] for enhanced performance. The main limitation of such approaches is to adopt a query-at-a-time paradigm, overlooking the potential benefits of reusing computations among queries that share common subexpressions. Tang et al. [11] look at benefit of sharing with varying deadlines for streaming queries. Their objective is to complete all queries, achieving 100% goodput, by allocating additional resources, without incorporating fault-tolerance mechanisms.

Altogether, in order to maximize goodput with fixed resources, current sharing systems lack awareness of deadlines, while existing deadline-conscious approaches overlook the importance of sharing. This creates a significant gap that needs to be bridged along with resiliency, which forms the objective of this work.

Challenges

Sharing: The primary objective of sharing-based systems is to maximize the sharing of common work to minimize the overall batch response time, even if it means sacrificing individual response times. This philosophy entails prioritizing long-term gains over short-term benefits. However, when deadlines are involved, this fundamental principle is put to the test, and conventional sharing methods may need to be reevaluated. In such situations, being mindful of deadlines is crucial when determining which queries to share and how best to share them to optimize performance.

Resiliency: Cluster-wide query processing is bound to encounter failures, which can pose significant challenges, particularly in meeting deadlines and in sharing contexts. Basic resiliency involves restarting queries on failures, which can be problematic if a failure occurs near the end of query execution or deadline. This results in very few queries being completed before the deadline. Advanced techniques, such as recovery from saved checkpoints [12–14], can be used to address this issue for individual queries, but they are sharing-oblivious and may limit the performance in batch scenarios.

Sharing and checkpointing techniques are closely related, and these independently explored methods may be tightly coupled. For example, consider a shared subexpression used by many queries, which, if not checkpointed, needs to be redone after a failure occurs. However, if this expression is checkpointed, the dependent queries have a higher chance of completion. Proactively managing failures within the context of sharing is essential for big data systems, given the importance of performance. In contrast, few techniques explore the allocation of additional resources to meet the given deadline. In our approach, however, we assume a fixed amount of resources, which is a typical use-case in practical settings.

1.1 Contributions

In this paper we present, BIGSHARED, a BIG data system for batch analytics with hard Deadlines via SHAREing. BIGSHARED is built on top of OpenLooKeng, an open source big data system that is a fork of Trino. Table 1 summarizes BIGSHARED's novelty in comparison to prior work, which are further enumerated in more detail as follows:

Prior Works	Sharing	Resiliency	Deadline
SWO [6]	✓	-	-
Datapath [15]	✓	-	-
SmartFaultTolerance [13]	-	✓	✓
XDB [14]	-	✓	-
RouLette [7]	✓	-	-
BIGSUB [5]	✓	-	-
iShare [11]	✓	-	✓
BIGSHARED **(this work)**	✓	✓	✓

Table 1: Taxonomy of Prior Works

(1) *Deadline-aware Batch Query Optimizer:* We propose a novel deadline-aware optimization algorithm for batch analytics in Section 3. The key idea is to partition the batch queries into *mini-batches* to aim for higher reuse and goodput. In order to achieve this, we introduce:
   - Query Shuffling for effectively partition queries based on subsumption,
   - Novel enhancements, such as *shared cost model*, over existing batch optimizer keeping it light weight and efficient,
   - A way to determine extent of sharing at run-time to keep up with deadlines.

(2) *Sharing-aware Checkpointing (Resiliency):* In Section 4, we devise a *sharing-aware checkpointing algorithm* that selectively checkpoints the most suitable subexpressions in order to maximize goodput. The selection prioritizes the subexpressions which are highly shared among batch queries and have lower *shared-cardinality*. This is the first work to address fault-tolerance in work-sharing.

(3) A novel attempt to implement shared execution in a distributed query-engine for ad-hoc workloads, unifying sharing, resiliency, and deadline. These involve non-trivial architectural designs, which are discussed in Section 5. Additionally, our changes are in compliance with the native system's architecture as it can support other engine functionalities in conjunction, which makes BIGSHARED beneficial from a deployment perspective.

We evaluate BIGSHARED on OpenLooKeng [16] using queries from TPCDS-benchmark. Our evaluation results are presented in Section 6 which shows that the goodput of batch of queries with BIGSHARED under failures is, on an average, 74% higher than state-of-the-art work sharing system, and 61% more than the optimal query-at-a-time approach.

2 BACKGROUND & PROBLEM

We first provide a background on the building blocks of shared query execution: (a) data query model, (b) shared operators, and (c) global query plan. Then we define the problem.

2.1 Data-Query Model

The Data-Query model [6, 15], as the name suggests, associates data (i.e. the tuples) with the corresponding queries. This is important since we are interested in *aggregated processing of multiple queries* on the *same data*. This is achieved by annotating each tuple with a *query-set information* that indicates the queries associated with the tuple at any stage of the execution. To elaborate,

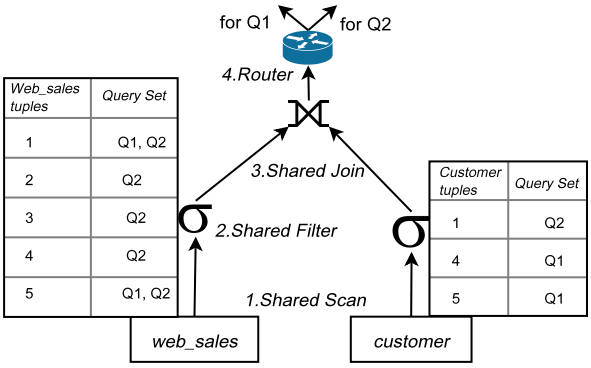

Figure 1: Data Query Model and Shared operators

Figure 1 depicts an example where queries Q1 and Q2 have joins over `web_sales` and `customer` tables with different filter predicates. For instance, the filter predicates in Q1, Q2 on web_sales are ws_quantity < 10, and ws_quantity < 20, respectively. Tuple id 1 to 5 of web_sales all satisfy Q2's filter predicate, while only tuple ids 1 and 5 are satisfied by Q1's filter. Further, since tuple 1 and 5 of web_sales are satisfied by Q1 and Q2, both are included in its query-set info. Since we append every tuple with this query set annotations, it is agnostic to the relationships among the filter predicates (subset, superset, etc.). Moreover, it automatically takes care of residual predicates by combining all filter predicates on a table in one scan.

2.2 Shared Operations

These operators, mooted in [6], are designed to share work among different queries. For instance, a scan operation on the same table and attributes (based on the storage model) can be shared across multiple queries. This is also depicted in Figure 1 with `web_sales` and `customer` tables.

First, sharing possibilities across queries are identified through shared filters as per the data-query model. Shared joins behave differently from traditional joins, wherein it additionally checks if the query set intersection of the joining tuples is not empty. For instance, the query set of the join of tuple id 1, is the intersection of query sets $\{Q1, Q2\} \cap \{Q2\}$, i.e. {Q2} goes through. While for tuple id 4 the query set intersection is null. An empty intersection means it is not required for both queries. Hence, the set intersection enables precision sharing, and also useful for preserving the data-query model after join.

Finally, a router operator is placed upon a shared subtree denoting the sharing of subexpression across multiple queries. Based on the query set information, router multicasts of shared operation to the associated parents. This is done to ensure that each record is forwarded to the appropriate operator/queries.

2.3 Global Query plans

The shared operators induce a plan for multiple queries which is referred to as the *global query plans*. Note that in the batch query setting with shared operators, instead of a single plan per query, we have a global plan for a batch of queries. Since a router can have multiple parents, a global plan is a directed acyclic graph of shared operators. An example global plan can be seen in Figure 3.

Optimizing global plans poses a challenge since the number of operators in such plans tends to be very large usually. For instance, the number of operators in a global plan can reach 100, even with a few tens queries in a batch. This becomes a bottleneck to the optimizer, which has a much larger search space compared to a single query scenario. Further, the problem is exacerbated with cardinality estimates for each operator as the cardinalities now correspond to shared cardinalities from multiple queries that can be erroneous with existing cost models. E.g. the selectivity of a shared operator is union of the selectivities of the participant operations.

2.4 Problem Definition

Our problem is as follows: given a deadline T and a batch of analytical queries, $\mathcal{B}$, the objective is to design a query processing technique that maximizes the number of queries completing within the time constraint T. In other words, the goal is to maximize goodput, where

$$goodput = \frac{\text{No. of. completed queries}}{T}$$

.

Note that all the queries in the input batch are submitted together, and each query has the same deadline. Handling different query deadlines and priorities is part of our future work.

3 DEADLINE-CONSCIOUS BATCH QUERY OPTIMIZER

In this section, we present our first contribution, which focuses on designing a batch query optimizer that is aware of deadlines. Simply sharing the entire batch of queries may not yield fruitful results when deadlines are involved, as it requires significant effort with initially low returns for long-term benefits. In some cases, the output of most queries may occur near the end of the shared batch query execution, while missing the deadline. Furthermore, also all the queries involved in a shared computation that fails before the deadline will also be unable to contribute to the goodput.

On the other hand, if we execute queries individually and miss a deadline, we would only lose the output of the last executed query. However, this approach overlooks the potential benefits of sharing work among queries. This realization serves as the motivation for the design of our solution, `BIGSHARED`. In this section, we present a simplified version of `BIGSHARED` that assumes no failures, and then then proceed to generalize the solution in the following section, where we account for the presence of failures.

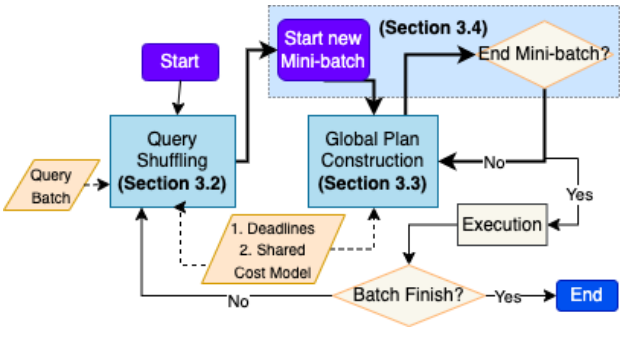

Figure 2: Deadline-aware Batch Optimizer Overview

3.1 Overview

The central idea involves achieving a balance between complete sharing *(full-share)* and no sharing *(no-share)* by dividing a large

batch of queries into smaller subsets termed *mini-batches*. Our approach, motivated by meeting deadlines, is detailed as follows:

(1) Effective Query Partitioning (Section 3.2): We establish a predefined query ordering (*query shuffling*) to group similar queries, maximizing common work reuse.
(2) Determining Mini-Batch Size (Section 3.4): This phase iterates over the queries in the *shuffled* order to construct a mini-batch. It terminates if the mini-batch's execution time is likely to exceed a deadline, as indicated by a cost model.
(3) Efficient Work Sharing (Section 3.3): This step defines a light-weight and efficient sharing strategy. It involves merging individual query plans to generate a *global query plan*.

This process is illustrated in Figure 2. Input queries are initially shuffled (query shuffling). The mini-batch creator iteratively selects unprocessed queries from the shuffled list to construct the global plan for the current mini-batch. This process continues until the end of the mini-batch is signaled. Then the global plan corresponding to the mini-batch is executed.

3.2 Query Shuffling

Intuition: As mentioned before, the idea is to find a query order that can be later used to find the mini-batch queries. This is based on the remaining deadline value at the start of the mini-batch. In order to maximize the goodput within the deadline, this component focuses on clustering similar queries together, and prioritize common shorter queries earlier in the order so as to achieve higher reuse. We leverage the concept of *subsumption* between queries to systematically determine the extent of common and reusable work among batch queries.

Q_i subsumes Q_j when the join expression of query Q_i is a subset of the join expression of query Q_j [1]. Figure 3 (a) provides an example, where $Q1$ subsumes $Q2$ since $Q1$'s join expression is a subset of $Q2$'s join expression. Similarly, $Q2$ subsumes $Q3$. By ordering Q_i before Q_j we achieve two benefits: a) the result of Q_i can be reused by Q_j, promoting efficiency; and b) a less costly query, Q_i (which subsumes Q_j) can be executed before a possibly more expensive Q_j. The opposite order (i.e., Q_j before Q_i) does not bring such advantages.

However, in practice, a query may not fully subsume another query, especially with a large batch of queries. For example, $Q3$ can only partially subsume $Q2$ since $Q2$ can possibly answer all $Q3$'s join expression except for join with table B. This necessitates the development of a general technique to handle subsumption.

Notations:

Join Graphs. Let $\mathcal{B}$ represent the batch of queries. Let us start with the well-known notion of *query join graph*. The query join graph, G_{Q_i} for each query $Q_i \in \mathcal{B}$, captures the join information in the query. There is a vertex for each table, and an edge between them if the corresponding join exists in the query.

Overlap Graph. Using the join graph, we now introduce *overlap graph*, $O(\mathcal{B}, E, W)$, with its vertices set being $\mathcal{B}$, and directed edges E between every pair of vertices. This graph formally captures the extent of subsumption or overlap of join expressions between queries. In O, with a vertex for each query, Q_i, and the edges in the overlap graph are assigned with weights, W, as follows: For each directed edge $Q_i \rightarrow Q_j$, its weight is given by its intersection cardinality over G_{Q_i}'s cardinality, i.e.,

[1]We do not consider filter expressions explicitly as they implicitly get shared with shared scan operator. More details in Section 2.1

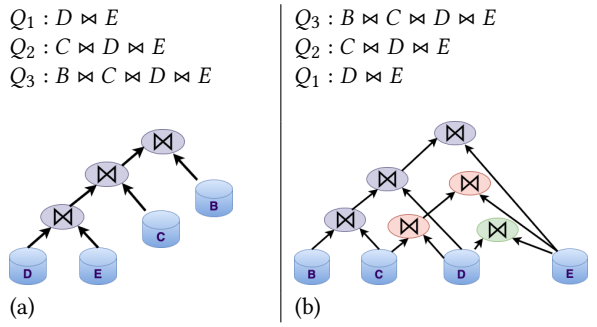

Figure 3: Datapath Optimizer and Limitations: (a) Q_1 subsumes Q_2, and Q_2 subsumes Q_3. The global plans with more reuse and sharing; (b) Global Plan with low sharing (not using subsumption property)

$$W(Q_i, Q_j) = \frac{|G_{Q_i} \cap G_{Q_j}|}{|G_{Q_i}|} \quad (1)$$

Here, any G_{Q_i}'s cardinality (and also intersection) is defined with respect to its edges. For the example batch of three queries, the overlap graph for $\mathcal{B} = \{Q_1, Q_2, Q_3\}$ along with the weights are shown in Figure 4. The higher the weight, imply the higher the overlap. Note that this weight function is similar to the Jaccard similarity measure [17].

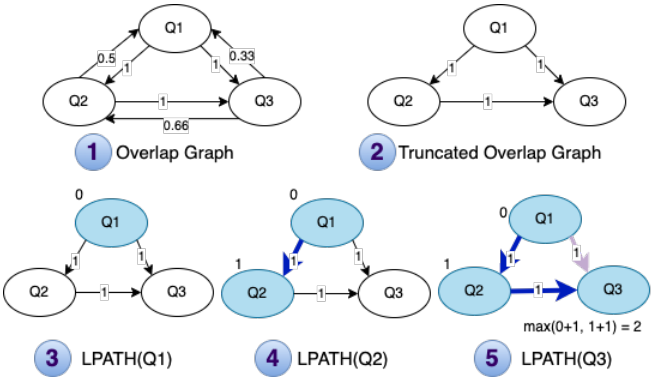

Figure 4: Overlap Graph and Query Reorder: Q1 → Q2 → Q3

Problem Formulation: To obtain the query shuffling order that maximizes subsumption, *a maximum weight path in the overlap graph needs to be identified where all vertices are visited.* This problem can be linked to finding the Hamiltonian path in the overlap graph with the highest weight. However, finding the Hamiltonian path is NP-hard, and therefore, the same is true for the weighted version.

Proposed Solution: Due to the inherent complexity of the problem, we propose an optimal solution by formulating it as an Integer Linear Program (ILP) and solving it using appropriate solvers. This approach is effective for relatively small batches or mini-batches, depending on the deadline values. However, it becomes computationally expensive with higher batch sizes. Therefore, we offer an efficient heuristics by transforming the graph into a directed acyclic graph (DAG) and devising a polynomial algorithm to find a path with maximum weight in the DAG.

This heuristic provides a reasonably good solution along with reduced computational overheads.

Optimal Algorithm. Let us now see the ILP formulation of the problem. Let x_{ij} be an indicator variable representing if an edge $i \rightarrow j$ in O is included or not in the maximum weight path [2]. Thus,

$$\begin{aligned} \text{maximize} \quad & \sum_{(i,j)\in E} w(i,j) * x_{ij} \\ \text{subject to} \quad & \sum_{i\in\mathcal{B}} x_{ij} = 1, \sum_{i\in\mathcal{B}} x_{ji} = 1 \quad \forall j \in \mathcal{B} \\ & \sum_{(i,j)\in E} x_{ij} = |\mathcal{B}| - 1, \quad x_{ij} \in \{0,1\}\ \forall (i,j) \in E \end{aligned}$$

The objective function is to find the maximum weight path, i.e, sum of weights of the selected edges. Each vertex must have one incoming and one outgoing edge (two constraints in the first line), ensuring each vertex is visited once. However, multiple subcycles or subtours can still be valid. So we add a constraint to limit the selected edges to $|\mathcal{B}| - 1$. This ILP can be solved with any of the typical solvers such as Z3 [18].

Heuristic Solution. The above optimal solution, however, can incur huge overheads with larger batch sizes based on the deadline values. Hence, we propose a heuristic with low overheads while possibly relaxing the solution quality. The key idea is to convert the overlap graph into a DAG by carefully pruning certain edges. Now the problem boils down to finding the *maximum weight path in a DAG*. There is an optimal solution for DAGs with $\theta(|\mathcal{B}||E|)$ time complexity, while the same is not true for general graph structures. The details are enumerated next:

(1) Given the overlap graph, first we construct a directed acyclic graph out of it. This is achieved by removing all the zero weight edges in the overlap graph. Further, in the remaining edges, retain the edge with a larger weight between any two vertices. For instance, given two vertices corresponding to queries Q_i and Q_j, we retain the directed edge $i \rightarrow j$ if the weight measure, as per Equation 1, of edge $i \rightarrow j$ is more than that of edge $j \rightarrow i$.
The resulting graph obtained after removing edges in this manner is referred to as the *truncated overlap graph*. The truncated overlap graph for the above-mentioned example is shown in Figure 4. Here, since Q1 subsumes all other queries, it has an outgoing edge to all other queries, but does not have any incoming edge.
(2) On the truncated overlap graph, we perform a topological sort to obtain initial vertex labels, disregarding weights. This labeling ensures that (i, j) is a directed edge in the overlap graph only when $i < j$.
Let $LPATH(j)$ denote the maximum weight path in the truncated overlap graph which ends at j. The following recursion is used to get the maximum weight path in the DAG:

$$LPATH(j) = \begin{cases} 0, \text{if } j \text{ is the source,} \\ \max_{i:(i,j)\text{edge}}\{w(i,j) + LPATH(i)\}, \text{otherwise.} \end{cases}$$

In order to prove the correctness of our proposed solution, we show that the truncated overlap graph is a DAG.

[2]For ease of presentation, using i interchangeably with Q_i

Lemma 3.1. *A truncated overlap graph is a directed acyclic graph.*

Proof. We need to show that the truncated overlap graph does not have directed cycles. Let us prove it by contradiction. Say that there is a directed cycle of size k, $V_1 \rightarrow \ldots \rightarrow V_k \rightarrow V_1$. It means that $\frac{|G_{V_1} \cap G_{V2}|}{|G_{V1}|} > \frac{|G_{V_1} \cap G_{V2}|}{|G_{V2}|}$. Since all the numbers are positive, it implies $|G_{V2}| > |G_{V1}|$. The same, when applied to each edge of the path, leads us to get $|G_{V_k}| > |G_{V_1}|$. However, from the final edge in the path, i.e, $V_k \rightarrow V_1$, we can conclude that $|G_{V_1}| > |G_{V_k}|$. This leads to a contradiction about the existence of a directed cycle, and hence the proof. □

In our batch example, this solution leads to the query order: $Q_1 \rightarrow Q_2 \rightarrow Q3$ (see Figure 4). Notably, longer-running queries are not necessarily delayed in the sharing process. The plan for Q3 is chosen in such a way that it reuses Q2 (hence, Q1 as well). Additionally, mini-batches often end with long-running queries that can be completed before the deadline.

3.3 Global Query Plan Construction

Intuition: The goal here is to create an efficient global plan for a mini-batch of queries. Inefficient plans struggle to exploit common expressions effectively (e.g., Datapath [15]). Moreover, optimizing complex global plans is challenging because they often involve DAG-structured plans with significantly larger operator trees (e.g., SWO [6]), leading to substantial overheads.

Our global plan optimizer is based on Datapath, known for its low overhead and incremental approach, ideal for mini-batch creation. We improve upon Datapath by adding two key elements: a) query shuffling as a preliminary step, and b) a novel shared-cost model during global plan construction. These enhancements yield higher-quality plans. Before delving into our novelties, let us briefly introduce Datapath next.

3.3.1 Datapath: In Datapath, a global plan is built iteratively for each incoming query within a fixed query order. For each join in the current query, it's added to the global plan if it can't be satisfied by the existing plan. Since the order of join checking within each query has an impact on the global plan quality, they resort to the A^* heuristic to find the minimal cost join order.

To illustrate this concept with a batch of three queries (Q1, Q2, and Q3), consider Figure 3 (a). The number of joins incrementally grows from Q1 to Q3, and these queries may share filter predicates through a shared operator. Starting with Q1, it gets the best individual plan, set as the current global plan. Then, for Q2, each of the two joins $(D \bowtie E)$ and $(C \bowtie D)$ are evaluated in order (assuming it as the order chosen by the A^* heuristic). It checks if $(D \bowtie E)$ join can already be satisfied by the current global plan (from Q1). Since it is satisfied here, the join is not added. While the same is not true with $(C \bowtie D)$ join, and hence, the corresponding join node is added to the global plan. This process continues until all joins in all queries are examined, resulting in a plan as in Figure 3(a).

Limitations. Datapath, though it is scalable, has a few limitations:

(1) *Query Order Sensitivity:* The order in which queries are given to the plan stitcher greatly affects global plan quality. For instance, reversing the order of a batch of 3 queries can lead to complex, suboptimal global plans (as shown in Figure 3 (b)), potentially the global plan having $O((\text{no. of joins})^2)$ operators in the worst case.

(2) *Shared Cost Model:* It lacks consideration for cardinality and cost estimates when shared operators are involved, which are distinct from traditional models.
(3) *Sub-optimal Intra-Query Ordering:* Exploration of join orders in an incoming query relies on the A^* heuristic, limiting the intra-query join ordering choices, impacting the plan quality.

3.3.2 Datapath Enhancements.

(1) **Query Shuffling based Ordering:** We use the ordering provided by query shuffling to build the global plan incrementally. This results in significantly higher quality plans as later queries inherently reuses the subexpression results of the latter ones. Figure 3 (a) captures the effect of shuffling based plan construction, resulting in high reuse and significantly less operators.
(2) **Cost Model for Shared Operators:** The key idea in our shared cardinality estimation module is to leverage the single query estimations and later combine them systematically. Specifically, when a shared join (or any shared operator) is shared among n queries, the selectivity of the shared operator can be estimated to the selectivity of the union of the n joins in these n queries. Let $\sigma(J_i)$ represent the selectivity of a single query operator J_i. Then, the selectivity of a shared operator across n queries with query q_i having operator J_i is represented by $\sigma(J_1 \cup \dots J_n)$. Using standard probability theory, we can infer: $\sigma(J_1 \cup \cdots \cup J_n) = 1 - \Pi_{i=1}^{i=n}(1 - \sigma(J_i))$. These new estimates are then plugged into the original engine cost model to get the new costs.

3.3.3 Intra-Query Join Ordering. Another limitation of Datapath is its reliance on a constrained search space for join orders, akin to the traditional join ordering problem in single-query scenarios. The optimal join order for an incoming query depends on selectivities and existing join nodes in the global plan.

In our solution, we exhaustively search all the possible join orders in the incoming queries. Note that the cost of a join can be very different from the single query setup, especially when some joins already exist in the current global plan. In such joins, the cost reflects the difference in cost incurred by the extra tuples processed by it. While this cost-based approach enhances plan quality, exhaustive search has significant time overhead. To limit optimization times, we employ dynamic programming-based join order searching with memoization.

Note that we consider only reordering the logical plan, though optimizing physical operators is also helpful. However, a joint logical-physical optimization is not practical as it is too complex. As commonly practised, we decouple the two phases: (1) first pass to decide on a logical plan, (2) always use a commonly used partitioned hash join (in big data systems) for the shared joins. In future, we can explore the merits of integrating other operator algorithms for shared joins.

3.4 Deadline Conscious Sharing: Mini-Batching

Intuition: Our approach, as discussed, initially processes queries following the order defined by query shuffling. We incrementally build the global plan, proceeding through the mini-batch until its end. Now, an interesting question arises: *How do we decide the mini-batch sizes?*

The key concept is to stop a mini-batch when the incoming query shares low or no common sub-expressions with the queries in the current mini-batch. In such cases, the incoming query becomes part of a new mini-batch. Additionally, if we anticipate that the new mini-batch with the incoming query will miss the deadline, we signal the end of the mini-batch.

Once a mini-batch is determined, it is sent for execution. After execution, the process repeats to determine the next mini-batch. This cycle continues until the entire batch is processed.

Algorithm: The following rules are used to determine the end of a mini-batch. Whichever rule is satisfied first becomes applicable:

(1) Terminate the current mini-batch when there are no outgoing edges in the overlap graph. As we traverse the queries in the query-shuffled order, a query with zero or low weight out-degree indicates that it does not or weakly subsume any other query. Therefore, it is preferable to start a new mini-batch.
(2) Stop the current mini-batch if the cost of the current global plan exceeds the cost equivalent of the remaining deadline value. This deadline pertains to the time left after earlier mini-batch completions, i.e., 'remaining deadline value' = 'original deadline value' - 'total execution times of all completed mini-batches'. To account for potential inaccuracies in cost models, we typically opt for conservative batch sizes, derived from observed runtime errors during runtime. Furthermore, we continually refine the cost model for accuracy.

Discussion: Note that Rule (1) is checked before Rule (2). If a query weakly subsumes another query, there is no point in checking Rule (2) as we anyway start a new batch. To determine its weak subsumption, we take the sum of weights of its outgoing edges and check whether it is below a threshold value (e.g.: < 0.15).

Given the potential for cost model errors (even after tuning the model as described next), the mini-batch cost may not precisely reflect the remaining deadline time. To mitigate this, we track the average error in the cost model and proceed conservatively, selecting smaller batch sizes to minimize cases where the mini-batch exceeds the deadline, thereby maximizing goodput. Monitoring cost model errors requires logging of query cost and execution time.

Tuning Cost Model: To enhance the accuracy of the engine's cost model, we tune the constants associated with CPU, IO, and network usage. OpenLooKeng cost model estimates the CPU, IO and network resources for each operator, which are then aggregated to determine the overall plan's resource usage and cost. The resource estimates are scaled by these constants to get normalized cost. Our tuning approach is along the lines of [19], where each of these constant factors are updated through the execution of specific calibrated queries tailored to each resource type.

4 GENERALIZATION WITH FAILURES

In this section we generalize our optimization solution with failures. This requires extending the cost model in face of failures and recovery. Further, we devise sharing-aware checkpointing and fine grained recovery to achieve performance benefits.

4.1 Failure Model

Considering different cluster wide processing scenarios, different types of failures may occur – ranging from process, node, operator, task and network failures. Our technique handles a variety of such failures which are typical in production environments.

Most of the prior works simplify recovery by assuming that if an operator fails, all instances of that operator across nodes also fail. However, we model *task-level recovery*, where only the failed tasks are recovered. This means that only tasks in the nodes affected by failures need to be restored, allowing for quicker recovery during node failures. Furthermore, our model assumes that the checkpointed results are not lost due to failures. To ensure this, all shuffle data (intermediate results) are materialized to a separate fault-tolerant storage medium (e.g., HDFS). In the event of failures, queries are restarted from the last successfully checkpointed intermediate result at a task level.

4.2 Background of Checkpointing Mechanisms

Query Restart: It is a naive resiliency mechanism, which manually restarts queries from the beginning with the hope of eventual completion. Restarting is done iteratively as many times as the failure happens. For batch setting, however, the longer the batch runtime, the more likely it is prone to such failures. This may result in low goodput due to lost work (across queries), especially when the failures happen near the deadline.

State Checkpointing: The concept involves pausing query execution at user-defined intervals and storing the current state of all running operators in persistent storage. In case of failures, the latest checkpoint is used for recovery. While this approach [13, 20] is attractive, it has the following limitations:

(1) Taking distributed snapshots is rather hard to implement.
(2) Snapshotting can be time-consuming with large task states and numerous workers. This is more challenging with batch scenarios where a plan tends to have hundreds of operators even with medium-sized query batches.
(3) Offers coarse granularity and limits the opportunities for adaptive query processing such as join reordering, operator selection and partition tuning.
(4) All tasks must restart from a checkpoint after a failure, which can be problematic if failures occur more frequently than snapshots, and if a high-cardinality operator needs checkpointing.

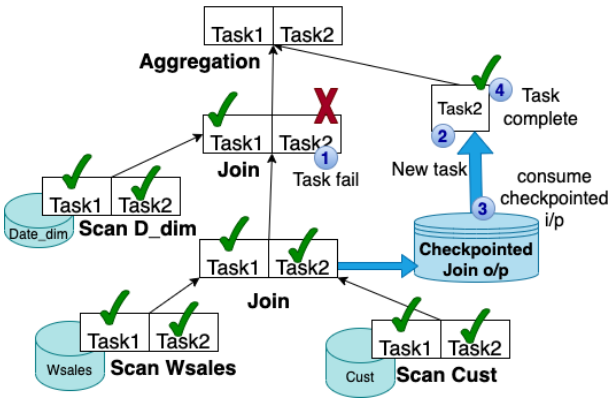

Figure 5: Selective Stage Checkpointing (exchange operator abstracted)

Stage checkpointing: The idea for this approach is to checkpoint at the granularity of stages [12, 13]. Outputs of stages (i.e. an end of pipeline in a distributed context) are stored in reliable storage. Figure 5 depicts an example scenario wherein, after the scan of `wsales` and `customer` tables, the data are *exchanged* (and is checkpointed) over the network (or locally) to other nodes. Further, after the joins, again the output is checkpointed, which is then passed to aggregate operators. This coarse stage-level checkpointing offers the following advantages, and hence we leverage it in this work:

(1) Most importantly, stage checkpointing does not break the query processing pipeline (which significantly impairs performance), unlike state or operator checkpointing.
(2) Stage checkpointing adds less overheads since even in normal operation (w/o checkpointing) either stage output is sent over the network or spilled to the disk in case the output size exceeds a threshold value. The end of stages are captured by an exchange operator (not shown in Figure 5).
(3) It allows efficient task-level recovery (a stage is composed of parallel tasks, each on different threads). Each task can be independently (in parallel) restarted from the saved checkpoint in case of failures. In our example, a task in worker 2 fails. For recovery, only this task is retried. It uses the stage checkpoint, the previous join stage output.
(4) Advanced techniques such as dynamic reoptimization can come into play here as queries can be scheduled partially and re-optimized at the stage boundaries.

In short, in stage-level checkpointing, all tasks in a stage perform checkpointing after completion, and only the failed tasks recover using it. For our POC implementation, a child stage starts only after all the tasks in the parent stage complete checkpointing. This design is very common in practice, such as in [12]. Making the stages asynchronous with respect to checkpointing is part of our future work.

4.3 Cost Estimation with Failures

An important component in our approach is to estimate the query cost in the face of failures. As we have seen before, the batch optimizer is dependent on the cost model to determine the mini-batch for subsequent iterations. We need to extend the existing cost models to capture the node failures and recovery of the tasks only for the failed nodes. To do so, we decompose the cost estimation of the whole query into estimating the cost of stages (and tasks) and combining them. Thus the query execution time, T_{query}, becomes:

$$T_{query} = Max_{\{stage\}}\{EndTime(stage)\} - Min_{\{stage\}}\{StartTime(stage)\} \quad (2)$$

The above formula essentially means that the execution time is the difference between the earliest start time of the first stage and the end time of the one that finishes last. Next, we look at the breakdown of stages into tasks with failures/recovery:

$$T_{stage} = Max_{\{tasks\}}(T_{nofailure}(task) + \sum_{failures}(recovery)) \quad (3)$$

Here, the first term is the time needed to process the task in case of no failures. The second term is the expected time spent in detecting failures, reassign failed tasks to other nodes until the start of execution of failed tasks. Finally, the maximum of these times is the time taken for the tasks in a stage. The above model captures the cost model with failures at the task level. Notably, it gives us the executor's times. The values for components with less variance, such as optimizer and scheduler, can be easily

estimated with high accuracy and can be added to the executor's times.

4.4 Sharing Conscious Checkpointing

Although the above approach is applicable to batch scenarios with shared operators, a significant portion of total batch execution time is spent on checkpointing. We further improve its performance with *selective checkpointing* by materializing only the *important* stages, so that it improves by saving on the IO while still achieving reasonable recovery times. The following two factors mainly play a role in identifying these important stages:

- *Dependencies*: If a stage is shared among many queries, then checkpointing it is beneficial. If we do not checkpoint such a stage output, then its failure would require redoing of all its dependent stages. This may cause a lower batch success rate.
- *Running time*: Success rate is also affected by the running time. Stages with low running time (i.e. easily recoverable) and fewer downstream dependencies are preferred to skip checkpointing.

We assign scores to all the above parameters for each stage and greedily pick until a cost budget (as function of deadline) is reached.

5 SYSTEM ARCHITECTURE

Figure 6 depicts the overall architecture of BIGSHARED. We now discuss some of the design as well as deployment aspects:

5.1 Batch Query Optimizer

BIGSHARED accepts a *batch* of queries along with a deadline as input. As explained in Figure 2, a loop of mini-batch planning is initiated after query shuffling (Section 3.2). Logical plans are constructed for each of the queries in the mini-batch (by optimizing them with the existing rules such as eliminating redundancy, predicate pushdown), which are then passed over to the *Global Plan Construction* (Section 3). All these individual plans are possibly reordered (Section 3.3.3) to maximize sharing, and stitched into a *shared plan or a global plan* till the mini-batching ends. After having the global plan constructed, we make two more passes over it before scheduling it for execution. The first pass adds router operators and the second pass *harmonizes* or matches the routers' inputs for the shared expression. During the first pass, we identify the subexpressions for which we should share execution. On top of each of them, we add a router for multicasting the output of shared operators. In the second pass, we union the projection attributes of all the shared expressions so as to provide a consistent interface for the routers. After this, the *selective checkpoint* (Section 5.3) adds stage-checkpoints to the global plans and then passes on the plan to the physical planner.

5.2 Shared Executor

Data-Query Model with Grouped-Filters: Grouped filters store predicates in an easily searchable data structure (e.g., search trees), that reduces filter evaluation into a traversal of sublinear time complexity. We build a grouped filter per attribute. The final filter operator output, representing the query-set satisfied by the input tuple, is determined by intersecting all the grouped filters together.

Shared Operators: The figure highlights shared operators in action (inside the Runtime). They allow us to share common work across queries. The common works are identified through the query set model. Our shared scan, filter and join operator maintains query-sets as an extra block (or column) that is transparently added to the output.

Router operators are judiciously placed by the optimizer to mark the end of a shared sub-expression. These routers multicast, by selectively transmitting data based on associated query-sets. They adhere to the producer-consumer model, where each shared subtree has a single producer router instance and a number of consumer instances equal to the downstream operators. Communication between the producer and consumers occurs through shared queues, with one dequeue per consumer. The producer router inserts incoming pages into all consumer dequeues. Upon dequeue, each consumer retrieves a page from its own dequeue and matches the query-set of each tuple with its assigned queries. The newly formed page is then routed for execution, with an exchange operator added immediately after the consumers for distributed execution. The overhead introduced by the router primarily stems from query-set operations during enqueuing, dequeuing, and exchanges.

As the consumers fetch data from these queues to pass on to downstream (shared or native) operators, all of them need to be executed on the same node. The producer and consumer routers belong to different stages. The scheduler has the responsibility of identifying a common worker node for such dependent stages.

5.3 Resiliency

Failure-Detection: BIGSHARED employs a time-out-based failure detection mechanism at the task level. Each task periodically sends status-update signals or messages to the coordinator. If the coordinator does not receive an update within a specified time frame (timeout), it assumes the task has failed. Subsequently, the coordinator node initiates the recovery process for the failed tasks. The overhead for task-level updates is minimal in big data settings, offering the added benefit of fine-grained recovery.

Fault-Tolerance: BIGSHARED ensures resilience through *stage checkpointing*, as explained in Section 4. In this process, intermediate data generated by a stage is stored in reliable storage like HDFS. This method offers substantial checkpointing savings compared to conventional operator-level approaches. Our approach to selective checkpointing focuses on materializing specific stage outputs to improve goodput while adhering to deadlines. The selection criteria include the stage's importance, measured by the number of dependent queries, and the cardinality of its output.

Scheduler: The scheduler organizes plan fragments into stages and executes them concurrently as tasks. Fragments are scheduled in a topological order, ensuring that independent stages finish before dependent ones. Throughout execution, certain stages create checkpoints as given by the plan. If a task fails, the scheduler instructs it to restore its states from the checkpoints, facilitating recovery and resuming execution. Tasks without checkpointed states recover through recomputation based on input data. As noted before, recovery does not occur instantly; there is a delay caused by the detection of the failure before the recovery process is triggered.

6 EXPERIMENTAL EVALUATION

In this section, we assess the effectiveness of BIGSHARED on a representative collection of complex OLAP queries, and conduct a comparative analysis against state-of-the-art techniques.

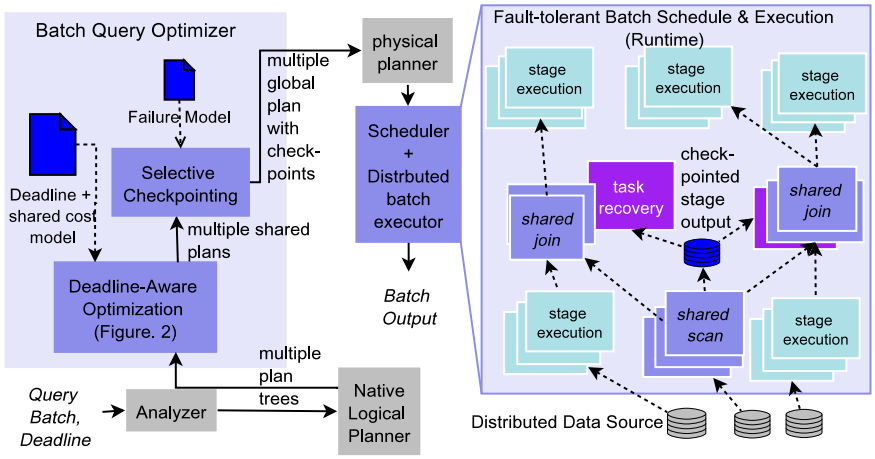

Figure 6: Architecture of BIGSHARED

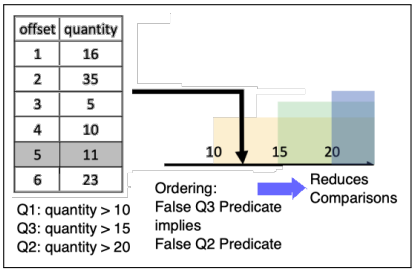

Figure 7: Group Filter

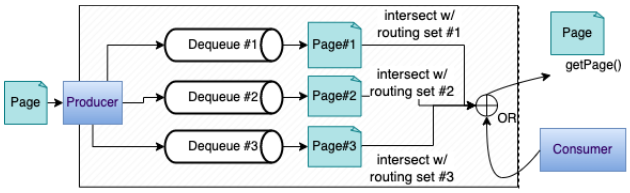

Figure 8: Router Producer Consumer Model

6.1 Experimental Setup

Database Engine: We have implemented BIGSHARED and the baseline approaches on OpenLooKeng [16] - an open-source big data system based on Trino [2] that is widely used in production environments. Within OpenLooKeng, the distributed query planning is performed by the coordinator node, while the plan fragments are concurrently executed by one or more worker nodes. Notably, OpenLooKeng follows the *storage compute separation* paradigm, enabling queries to be executed over diverse data sources, such as Hive, PostgreSQL and more. We use HDFS for stage checkpointing.

Hardware: We ran our experiments on a cluster of 4 identical machines with Intel(R) Xeon Gold CPU with 2.60GHz, 32GB RAM, and Ubuntu 20.04 Operating System.

Batch Workload: Our test workload is comprised of representative SPJA sub-queries from the TPC-DS benchmark with a scale factor of 10. In the absence of missing standard benchmarks for query batches, we create three types of diverse workloads of 100 queries each from TPC-DS capturing challenging scenarios.

(1) *Low-Share (W1):* Queries of the form A join B join C with a common fact table. Dimension tables and filter predicates are varied to generate the batch of queries. This workload evaluates our system in cases where only a few joins exist and the degree of sharing is low. This is hard case for BIGSHARED and Datapath.
(2) *High-Share (W2):* All queries of this workload share a common template with 4 joins, with varying filtering predicates. These are scenarios where there is high overlap among queries.
(3) *Regular-Share (W3):* This workload is a mix of high and low sharing queries with varying number of joins (2-5) and filter predicates on the fact table. It represents a typical use case.

Note that in each workload there is a common fact table store_sales (ss). It is the most commonly used fact table in TPC-DS. We pick 10 distinct SPJA sub-query from the benchmark containing store_sales - namely, Q1, Q7, Q38, Q42, Q43, Q55, Q73, Q79, Q87, Q96. We created six more query templates containing 4 and 5 joins required for Regular-Share using different combinations of the following dimension tables: store (s), item (i), date_dim (d), customer (c), income_band (ib) and household_demographics (hd). These query templates were created because some of the larger joins from TPC-DS (involving store_sales) were not supported in our current implementation (although extendable)– mainly, requiring no repeated tables and numeric filter predicates. The batch queries for the three workloads were then created from the above query templates by varying the filter constants along the numerical attributes.

Baselines: The performance evaluation of BIGSHARED includes a comparison against the following baselines:

- **Query-at-a-time approach (*QAT*):** This baseline represents the traditional execution model, where each query is processed individually.
- **Optimal QAT technique with deadlines (*QAT-OPT*):** Here, the query is rescheduled based on the shortest job first ordering, aiming for optimal execution within given deadlines. Note that we use engine's cost model for rescheduling [10]. This baseline also serves to assess the impact of trivially executing all the shorter running queries first for maximized goodput.
- **State-of-the-art sharing system (*Datapath*):** We observed that Datapath often shows degraded performance when sharing fully due to the bottleneck at the router operator. Hence, the whole batch is divided into mini-batches, of constant size based on the arrival order, for a fair comparison. Notably, we exclude the comparison against SWO [6] due to its limited scalability. SWO could handle a maximum batch size of 11 within a 1-hour timeout [7]. Non-intrusive approaches such as [5] are not suitable for ad-hoc query batches that differ from workload history.

To ensure fair comparison and fault-tolerance, we employ stage checkpointing across all the techniques. Stage-checkpointing strikes a balance between no-materialization, which is inefficient, and full materialization of all operators, which is resource-intensive. The superiority of stage-checkpointing over both full

and no materialization has been noted in [13]. For further details, refer to Section 4.

Metric: We measure the performance of each technique by Goodput, i.e. the query completion rate at deadline T for an input batch. The results are normalized to QAT performance.

Deadlines: When the deadlines are too high or too low, the goodput achieved by various techniques becomes comparable. In order to establish a benchmark performance, we select the deadline value such that around 1/3rd of the batch queries are completed with QAT (for single deadline evaluations).

6.2 Performance Comparison: No Failure

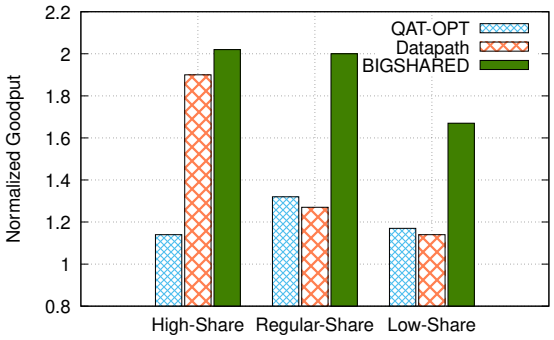

Figure 9: Comparison of goodput with batch size of 100 and deadline of 300 secs (No failures)

In this evaluation, we compare the goodput rates of different techniques in the absence of failures across Low-Share, High-Share, and Regular-Share workloads. The objective is to assess the effectiveness of BIGSHARED in highly reliable environments where failures are rare occurrences. All workloads are assigned a deadline of 300 ms. By default, for selective materialization, BIGSHARED runs on a cost budget corresponding to 25% of the chosen deadline value.

Figure 9 demonstrates that BIGSHARED outperforms other techniques consistently across all workloads. On average, it surpasses Datapath (pure sharing) by more than 1.5 times in both Low-Share and Regular-Share scenarios. It is noteworthy that under the High-Share workload, Datapath and BIGSHARED exhibit similar success rates as they behave similarly in context of query shuffling. However, BIGSHARED gains an edge by incorporating deadline awareness to the process.

Both QAT-OPT and Datapath exhibit comparable performance in Low-Share and Regular-Share workloads, but not for High-Share. High-Share represents the most favourable condition for sharing systems. However, QAT-OPT falls behind BIGSHARED by missing out on the savings achieved through sharing. The gap between the two techniques is more pronounced in High-Share and Regular-Share scenarios compared to Low-Share. Even for Low-Share, where there are fewer joins in common, BIGSHARED leverages sharing of work from scanning common tables, grouping filter operators together across queries. In effect, BIGSHARED outperforms QAT-OPT by 43%.

Overall, BIGSHARED significantly improves goodput compared to the best query-at-a-time paradigm and the state-of-the-art sharing system, achieving performance enhancements of up to 78% and 57%, respectively.

6.3 Performance Comparison: With Failures

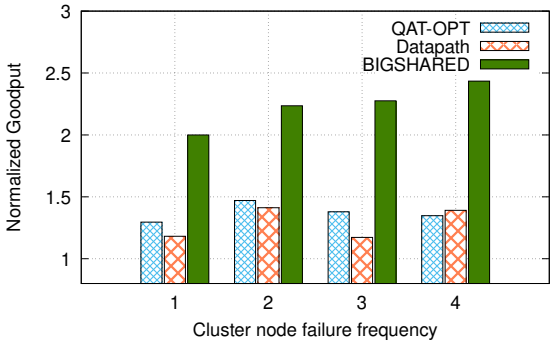

Figure 10: Comparison of Goodput with varying failures and deadline of 400 secs

Now, we assess the performance of all the techniques in case of failures. We induce failures in the workers by killing one worker [3] at a time at regular intervals of 100 secs. Since there are failures that induce delay, we set the deadline to a higher value of 400 secs. Figure 10 shows the goodput with varying number of failures of all the techniques on the regular share workload. Note the recovery is at a fine grained task level, i.e., only the failed tasks of a stage are retried for recovery. Note that the failure detection timeout is set to 1 minute.

QAT-OPT shows relatively consistent performance across failures, as the impact of a failure is limited to the query running at that particular time. On the other hand, sharing-based approaches display more variance as they share and consequently lose more work in the presence of failures. The performance of BIGSHARED, however, is better safeguarded due to its strategic use of checkpoints in critical stages, coupled with deadline-aware sharing. Comparing BIGSHARED to QAT-OPT, the former achieves a 55% improvement in goodput with one failure, which further increases to 81% with four failures (similar comparative performance w.r.t. Datapath too). As noted earlier goodput numbers are normalised to QAT performance. Since QAT performance deteriorates at a faster rate compared to BIGSHARED, it leads to an increase in relative performance – thanks to deadline-aware optimization, selective checkpointing and careful sharing leveraged by BIGSHARED. As observed, the oscillation in the deadline-oblivious Datapath is because performance depends on whether the current mini-batch finishes before the deadline. Finally, the failures affect QAT-OPT in a similar manner as QAT.

Overall, even in presence of failures, BIGSHARED outperforms other techniques across all workloads, surpassing QAT-OPT by 61% and Datapath by 74%, respectively, on average. This showcases the superior resiliency of BIGSHARED compared to alternative approaches on multiple failures.

6.4 Optimization Times

Figure 11 captures the cumulative optimization times of the baselines and BIGSHARED on the regular share batch of 100 queries. Dynamic-programming based intra-query ordering is applied to both Datapath and BIGSHARED. Memoization helps to get better quality plans without sacrificing on the optimization times. Further, Datapath and BIGSHARED have similar optimization times,

[3]The worker comes back again after a few seconds, resulting in a 3-node cluster at each failure point

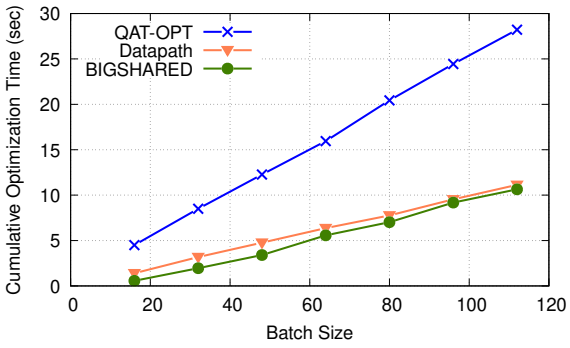

Figure 11: Comparison of Optimization times

while being much faster than QAT-OPT. This is because the optimization time for QAT-OPT includes query parsing, logical and physical planning for all batch queries. In sharing approaches, query parsing times remain the same as QAT-OPT. However, logical and physical planning are performed once per mini-batch – this component is just 3X higher (for mini-batch size=16) compared to a single query.

The additional query shuffling phase is there in BIGSHARED but not in Datapath. However, it is compensated with less optimization overheads of global plans for mini-batches in BIGSHARED. This is because mini-batches have more query similarity in BIGSHARED in comparison to Datapath – thanks to query shuffling – leading to reduced search space, and hence, less complex plans.

6.5 Varying Deadlines

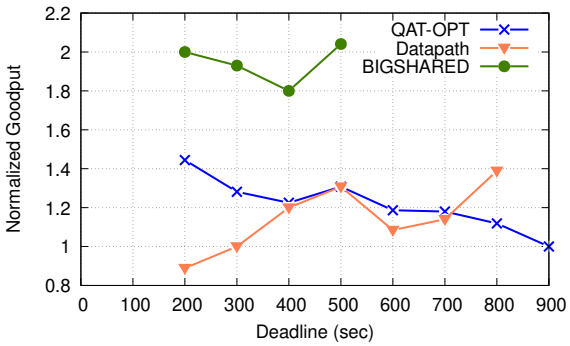

Figure 12: Varying Deadlines

In the course of this evaluation, we systematically assessed by altering the deadline duration within a range spanning from 200 seconds to 900 seconds, and measuring the goodput of all the techniques. Note that there are different runs of each technique for each deadline value. The results of this analysis are presented in Figure 12. The evaluation was performed on a Regular-Share batch that included three failures. To ensure an even distribution of failures across the runtime of both Datapath and QAT-OPT, the failures were intentionally induced at intervals of 200 secs.

Among the evaluated techniques, BIGSHARED achieves the fastest completion of the entire batch, finishing in 500 secs (indicated by line termination) compared to the 800 to 900 secs required by QAT-OPT and Datapath. As QAT-OPT prioritizes executing the shortest jobs first, it exhibits higher goodput compared to Datapath. However, with large deadline values, sharing becomes inherently advantageous, and QAT-OPT executes larger queries at later stages.

Despite performing well with short deadline values, QAT-OPT still falls short compared to the performance of BIGSHARED due to its lack of work sharing capabilities. In contrast, BIGSHARED not only prioritizes shorter queries to a certain extent, as explained in the query shuffling process, but also incorporates sharing of work. There is a noticeable dip in the performance of BIGSHARED around the 400-second mark, which can be attributed to a failure that occurred before that point. When the system recovered, with around 40 secs remaining to run a mini-batch, it chooses a small mini-batch of size 4 for execution.

Overall, BIGSHARED demonstrates strong performance across the entire spectrum of deadline values, finishing the entire batch 63% earlier than Datapath. This impressive efficiency is achieved by combining the strengths of QAT and sharing approaches.

6.6 Impact of Dynamic Mini-Batching

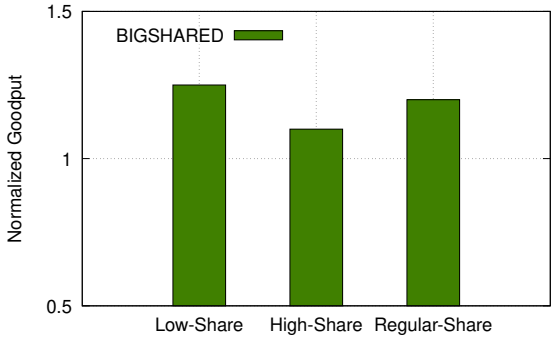

Figure 13: Effect of Dynamic Mini-Batching

Let us now assess the impact of our proposed approach in Figure 13. We consider all three workload types while keeping the number of failures fixed at 3 and setting a deadline of 400 seconds. The y-axis represents goodput values for BIGSHARED, normalized to BIGSHARED without dynamic mini-batching.

It is evident that Low-share and Regular-Share workloads outperform High-Share workload on goodput. This disparity arises because High-Share had relatively little time after recovery with deadline following up. For other workloads, without dynamic mini-batching, a substantial amount of work was lost due to missed deadlines. However, with dynamic mini-batching, a larger portion of the mini-batch work could be accomplished. In summary, dynamic mini-batching enhances the robustness of BIGSHARED performance concerning the timing of failures.

6.7 Discussion

While BIGSHARED shows benefit for various analytical workloads, it may have limited utility in the following scenarios:

- *Low-Latency Queries (LLQs):* Even the optimization phase is often skipped for LLQs for faster response. Resiliency is relevant for medium to long-running queries, as re-computation is costly. Recovery from failure itself takes seconds, which involves reading of materialized data, slowing down the normal execution. Therefore, LLQs cannot gain benefit from BIGSHARED.
- *Low Selectivity Queries:* Sharing advantages might be limited in queries with very low selectivity or queries having less commonality as the scope of reuse is less.

7 RELATED WORK

The prior-art to our work spans three dimensions: work sharing systems, real-time databases and resiliency, as summarized next.

7.1 Work Sharing Systems

Numerous studies delve into Multi Query Optimization (MQO), starting with Sellis' seminal work [21]. These works detect common subexpressions among queries, reducing computation redundancy [9]. This concept expands in [22] and [23], and considers a wider context of map-reduce in [24, 25]. However, MQO's expensive subexpression matching limits scalability to a few tens of queries, as noted in [6].

Shared Work (SW) systems deviate from traditional MQO, aiming to share work across queries. These systems share operators on the same table or attributes and even joins, minimizing subexpression matching. Examples include Crescando [26], sharing scans among hundreds of queries, and more systems like Datapath [15], CJoin [27], QPipe [28], and SharedDB [29]. Finally, MQJoin extends ShareDB[30] with high-throughput joins.

Some approaches optimize global plans, such as Shared-workload Optimization (SWO) [6] and Datapath [15]. SWO frames the problem as a bi-linear optimization problem but does not scale well [7]. Datapath offers a scalable alternative for batch optimization and serves as the foundation for BIGSHARED optimizer (Section 3). Recent work like RouLette [7] addresses scalability through adaptive query processing, but necessitates a complete engine redesign, limiting its adaptability. Finally, SW systems can benefit from leveraging materialized views [5, 31], which can be used alongside BIGSHARED for enhanced performance.

7.2 Real-Time Databases

A substantial body of literature on real-time databases focuses on imposing real-time constraints based on application requirements. These systems are typically evaluated by analyzing missed transaction frequency and lateness in meeting deadlines, while ensuring data consistency. Various aspects are explored, including transaction scheduling, real-time concurrency control, buffer management, overload management, and distributed real-time databases [10, 32].

Transaction scheduling is particularly relevant to BIGSHARED in batch-analytics context. When deadline values are the same, the shortest job first algorithm is optimal. However, for tasks with different deadlines, algorithms like most-critical-first, earliest-deadline-first and their variations [33] are established. These algorithms consider different deadline values for each task [10].

Additionally, there are generalizations of these algorithms for online scheduling, where all tasks are not known upfront. Variants have been proposed for different types of deadlines, such as hard, soft, and firm. However, BIGSHARED considers a simpler setting with the same *hard* deadline values for all queries, equal priorities, offline mode, and similar conditions. Finally, Tang et al. [11] employ sharing through materialized views in streaming queries with varying deadlines. They aim to complete all queries with higher throughput using materialized views, without resiliency. In contrast, we consider fixed resources and resiliency, balancing sharing with deadline. In short, all these earlier works have seen queries in isolation and missed opportunities from sharing work. Furthermore, these algorithms assume an accurate estimate of task completion times, which is rarely true in practice.

7.3 Resiliency

Popular MPP data engines such as Teradata [34], Greenplum [3], Vertica [35], Impala [36], and HAWQ [37] retry parts or the entire query until it succeeds, to handle failures. Traditional databases employ techniques like storage-level replication [38–40] or checkpointing intermediate results [41]. Spark [1] uses RDDs, to capture data lineage, for fault tolerance.

Single-query fault tolerance solutions include FTOpt [42] and XDB [14]. XDB uses a cost-based approach to choose which operator to checkpoint, but breaks the processing pipeline and leads to longer recovery times when an operator fails. In contrast, stage checkpointing approach in BIGSHARED does not break the pipeline and recovers only the failed tasks. FTOpt offers an extensible operator level fault tolerance framework but relies on impractical assumptions like order-preserving data transmission between operators. Neither XDB nor FTOpt is designed for MPP engines. Stage checkpointing has been adopted in SmartFaultTolerance [13] and Trino [12], but has not been devised for batch scenarios. Moreover, SmartFaultTolerance looks at producing a plan with highest success probability. However, the approach does not scale well for large plans as it relies on Monte Carlo probability estimation to decide the plans, which becomes a huge bottleneck with large global plans as in our case.

In summary, existing resiliency approaches have focused on a query-at-a-time paradigm. This work re-evaluates them in the context of batch queries with a sharing-conscious perspective.

8 CONCLUSION

In this paper, we introduced BIGSHARED, designed to optimize goodput by efficiently sharing workloads among a batch of queries while considering deadlines. Our batch query optimizer, tailored for deadline-sensitive scenarios, manages the delicate balance between meeting deadlines and leveraging sharing, even in the presence of failures. We have incorporated selective stage-level checkpointing to enhance overall system performance further. BIGSHARED represents a significant advancement in enhancing system goodput through efficient sharing.

In the realm of real-time databases, various problem formulations related to deadlines have been explored, including online versions, deadlines with slack, queries with priorities, and deadline hardness. All of these, in the context of BIGSHARED, present interesting directions for future research. Equally important are tightly integrated engine components, such as the scheduler and buffer management, for improved performance. Also it will be interesting to explore leveraging GPUs for optimizing large global plans along the lines of [43, 44].

Exploring the applicability of BIGSHARED's approach in other domains, such as cross-geographical analytics, holds promise for future research. Some of the challenges we anticipate include managing slow and variable WAN bandwidth, new execution operators, optimizing operator (and router) placements, and scheduling them, among others.

REFERENCES

[1] "Improved fault-tolerance and zero data loss in apache spark streaming." [Online]. Available: https://www.databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html

[2] "Trino: a query engine that runs at ludicrous speed." [Online]. Available: https://trino.io/

[3] "Greenplum: Massively parallel postgres for analytics." [Online]. Available: https://greenplum.org/

[4] "Databricks." [Online]. Available: https://www.databricks.com/

[5] A. Jindal, K. Karanasos, S. Rao, and H. Patel, "Selecting subexpressions to materialize at datacenter scale," *Proceedings of the VLDB Endowment*, vol. 11, no. 7, pp. 800–812, 2018.

[6] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann, "Shared workload optimization," *Proceedings of the VLDB Endowment*, vol. 7, no. 6, pp. 429–440, 2014.

[7] P. Sioulas and A. Ailamaki, "Scalable multi-query execution using reinforcement learning," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1651–1663.

[8] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, "Computation reuse in analytics job service at microsoft," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 191–203.

[9] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 249–260.

[10] S. A. Aldarmi, , and A. B. A. 1998), "Real-time database systems: Concepts and design," Department of Computer Science, University of York, Tech. Rep. YCS-303, 1998.

[11] D. Tang, Z. Shang, W. W. Ma, A. J. Elmore, and S. Krishnan, "Resource-efficient shared query execution via exploiting time slackness," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1797–1810.

[12] "Trino: Fault-tolerant execution." [Online]. Available: https://trino.io/docs/current/admin/fault-tolerant-execution.html

[13] Y. Ji, Y. Chai, X. Zhou, L. Ren, and Y. Qin, "Smart intra-query fault tolerance for massive parallel processing databases," *Data Science and Engineering*, vol. 5, no. 1, pp. 65–79, 2020.

[14] A. Salama, C. Binnig, T. Kraska, and E. Zamanian, "Cost-based fault-tolerance for parallel data processing," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 285–297.

[15] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The datapath system: a data-centric analytic processing engine for large data warehouses," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 519–530.

[16] "Openlookeng: A distributed, low latency, reliable data engine for all data." [Online]. Available: https://openlookeng.io/en/

[17] L. J, R. A, and U. J, *Mining of Massive Datasets*. Cambridge, p. 74.

[18] "Z3 theorem prover." [Online]. Available: https://github.com/Z3Prover/z3/

[19] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton, "Predicting query execution time: Are optimizer cost models really unusable?" in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 1081–1092.

[20] "Operator snapshot: openlookeng documentation." [Online]. Available: https://docs.openlookeng.io/en/docs/docs/admin/reliable-query/operator-snapshot.html

[21] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 1, pp. 23–52, 1988.

[22] T. Kathuria and S. Sudarshan, "Efficient and provable multi-query optimization," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2017, pp. 53–67.

[23] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 533–544.

[24] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: Sharing across multiple queries in mapreduce," *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 494–505, sep 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920906

[25] G. Wang and C.-Y. Chan, "Multi-query optimization in mapreduce framework," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 145–156, nov 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732234

[26] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable performance for unpredictable workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 706–717, aug 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687707

[27] G. Candea, N. Polyzotis, and R. Vingralek, "A scalable, predictable join operator for highly concurrent data warehouses," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 277–288, aug 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687659

[28] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 285–291. [Online]. Available: https://doi.org/10.1145/3359989.3365433

[29] G. Giannikis, G. Alonso, and D. Kossmann, "Shareddb: Killing one thousand queries with one stone," *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 526–537, Feb 2012.

[30] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann, "Mqjoin: Efficient shared execution of main-memory joins," ETH Zurich, Tech. Rep., 2016.

[31] S. Liu, B. Song, S. Gangam, L. Lo, and K. Elmeleegy, "Kodiak: Leveraging materialized views for very low-latency analytics over high-dimensional web-scale data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1269–1280, 2016.

[32] B. Kao and H. Garcia-Molina, "An overview of real-time database systems," *Real Time Computing*, pp. 261–282, 1994.

[33] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest deadline scheduling for real-time database systems," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1991.

[34] "Teradata." [Online]. Available: https://www.teradata.com/

[35] "Vertica." [Online]. Available: https://www.vertica.com/

[36] "Apache impala." [Online]. Available: https://impala.apache.org/

[37] "Apache hawq." [Online]. Available: http://hawq.incubator.apache.org/

[38] "High availability and scalability guide for db2 on linux, unix and windows." [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247363.pdf

[39] "Oracle data guard." [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14239.pdf

[40] "Microsoft. high availability solutions (sql server)." [Online]. Available: https://technet.microsoft.com/en-us/library/ms190202(v=sql.110).aspx

[41] H.-I. Hsiao and D. DeWitt, "A performance study of three high availability data replication strategies," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 1991, pp. 18–28.

[42] P. Upadhyaya, Y. Kwon, and M. Balazinska, "A latency and fault-tolerance optimizer for online parallel query plans," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 241–252. [Online]. Available: https://doi.org/10.1145/1989323.1989350

[43] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki, "Efficient massively parallel join optimization for large queries," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 122–135.

[44] V. Mageirakos, R. Mancini, S. Karthik, B. Chandra, and A. Ailamaki, "Efficient gpu-accelerated join optimization for complex queries," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 3190–3193.